1) Consider the following design for a restricted access container (RAC):

```
// IRACContainer
+ void (clear())
+ boolean (isFull())
+ List (elements)[]
+ Object (get())
+ void (put(Object reql))
+ Object (peek())

ALRIRACContainer
+ IRALgo (strategy())

PQComparatorIRACFactory
RandomIRACFactory
LRSQueueIRACFactory
LRSSkipIRACFactory
```

This is very similar to the RAC presented in class except for one very important difference: the 

IRACContainer.isEmpty() method is absent.

A RAC behaves differently when it is empty or non-empty. Therefore, it makes sense for a RAC (IRACContainer) to be able to accept a two-case visitor.

a) Write the signatures for the methods of a visitor to the above RAC and the signature the method that should be added as a result to IRACContainer:

```java
public interface IRACVisitor {

    public Object emptyCase (IRACContainer host, Object param);
    public Object nonEmptyCase (IRACContainer host, Object param);
}

public interface IRACContainer {

    // … other methods not shown…

    public Object execute(IRACVisitor v, Object param);
}
```

1 of 9
b) Write the code for the method of LRSRAContainer (ALRSF$LRSRAContainer in the above UML class diagram) that implements the signature you added to IRAContainer.

Tip: To access the outer LRSRAContainer object from within an anonymous inner class inside a LRSRAContainer, use the syntax LRSRAContainer.this.

NO CONDITIONALS ARE ALLOWED IN YOUR SOLUTION! Think simply and directly!

```java
public Object execute(final IRACVisitor v, final Object param) {
    return _ls.execute(new IAlgo() {
        public Object emptyCase(LRStruct host, Object nu) {
            return v.emptyCase(LRSRAContainer.this, param);
        }
        public Object nonEmptyCase(LRStruct host, Object nu) {
            return v.nonEmptyCase(LRSRAContainer.this, param);
        }
    }, null);
}
```
2) In the Hangman project we represented a word whose characters were either visible or invisible as a modified list:

This model has several problems however:

- An NEWord in its invisible state does not actually represent an invisible word, but rather an invisible first. The rest of the NEWord could be either visible or invisible. Conversely, the same is true if the NEWord is in its visible state.
- The state pattern used in NEWord tightly couples the visibility property to the composite property of the word, even though those two properties really have nothing to do with each other.

So, what we would like to do is to replace IWord with IList, our immutable list framework. Thus, a Hangman word would be represented as a IList of WordChar objects (that you will design).
Design a model that represents the characters held by the `IList`. This model should include the class `WordChar` that an `IList` holds to represent the word in Hangman. Be sure to replicate all the non-list functionality of the `IWord` system including the ability to run visibility-dependent algorithms, e.g. `ToString`, `IsVisible`, `GuessChar`, etc.

Draw the UML class diagram of your model below. You do not need to write any code.

**Solution:**

```plaintext
draw the UML class diagram of your model below. You do not need to write any code.
```
3) Write a LRStruct visitor called Unzip to break up the host LRStruct into two lists. One list contains all the even indexed elements of the original host list; the other list contains all the odd-indexed elements of the original host list. We want the host to become the LRStruct that contains only its original even-indexed elements, and the returned Object of the visitor’s methods to be an LRStruct holding the original odd-indexed elements.

```java
public class Unzip implements IAlgo {
    public final static Unzip Singleton = new Unzip();
    private Unzip() {
    }

    public Object emptyCase(LRStruct host, Object nu) {
        return new LRStruct();
    }

    /**
     * @param host is not empty
     * @param nu not used
     * @return LRStruct containing the odd indexed element of the original host.
     */
    public Object nonEmptyCase(LRStruct host, Object nu) {
        // Hold on to the tail before setting the host's rest to a different list:
        LRStruct tail = host.getRest();
        // Get the odd indexed elements of the tail:
        LRStruct other = (LRStruct) tail.execute(this, null);
        // The odd indexed elements of the tail are the even-indexed elements
        // of the original host:
        host.setRest(other);
        return tail;
    }
}
```
The `findIndex(...)` method of the array-based dictionary given in lecture #28 uses binary search to look for a given key (see code below). Rewrite `findIndex(...)` using recursion only (no iterative loops such as for loops and while loops are allowed). Note: you may need to add a helper method.

```java
private DictionaryPair[] _pairs = new DictionaryPair[1];
private int _firstEmptyPair = 0; // # of elements in _pairs which is initially empty

// Other IDictionary methods have been omitted for clarity.
/**
 * Returns either (1) the index of the key if it is stored
 * in the array or (2) the index of the next smaller key
 * if it is NOT stored in the array. (If there is NOT
 * a smaller key in the array, returns -1.)
 */
private int findIndex(Comparable key) {
  int lo = -1; // Note that lo is one less than the first element of _pairs to be considered!
  int hi = _firstEmptyPair; // Note that hi is one more than the last element of _pairs to be considered!

  while (lo + 1 != hi) {
    int mid = (lo + hi) / 2;
    int result = _pairs[mid].getKey().compareTo(key);

    if (result > 0) // _pairs[mid].getKey() > key
      hi = mid;
    else if (result == 0) // _pairs[mid].getKey() == key
      return mid;
    else // _pairs[mid].getKey() < key
      lo = mid;
  }
  return lo;
}

Solution:

```
5) An infix arithmetic expression is an expression where the binary operator is positioned in between its two operands as in $1 + 2$, for example. A postfix arithmetic expression is an expression where the binary operator is positioned after its two operands as in $1 2 +$, for example. By convention, the multiplication and the division operator have higher precedence over the addition and subtraction operators. The infix expression $1 + 2 \times 3$ corresponds to the postfix expression $1 2 3 * +$. The following algorithm to convert an infix expression to its equivalent postfix expression is due to Dijkstra.

a) Assume the input text is an infix expression containing for integers and operators $+,-,\times $ and $/$ and ending with a ‘$’ to mark the end of the input. Think of $ as an operator whose precedence is lower than $+, -, \times $ and $. Use a stack S to perform the following steps:

b) Push a ‘$’ Character object onto S.

c) Get the next token from the input.

d) While the input token is not $ and not at the end-of-file do the following:

i) If the input token is a number then output it,

ii) else if the input token is an operator then pop and output all the elements in the stack S that have higher precedence than the current operator input token and stop as soon as the fist operator that has lower precedence appear at the top of S (do not remove it!). Push the current input token (as a Character object) onto S.

iii) else throw an exception because the input contains unknown tokens.

iv) Get the next token from the input.

e) Pop and output the rest of the stack S until the stack is empty.

Try the above algorithm by hand on the inputs $1 + 2 \times 3$ and $1 * 2 + 3$ to see how the above works.

Assume you have a method called boolean isHigherPrecedence(char op1, char op2) that takes two operators and returns true if op1 has higher or equal precedence than op2.

Design and write a method called infix2postfix(.) that implements the above algorithm to translate an infix expression from a given input Reader and output the equivalent postfix expression to an output character stream (Printer).

You may use the attached code (see the next page) for tokenizing in any way you want. When the tokenizer picks up $+, -, \times $ and $/$, the nextToken() method returns the ASCII code for $+, -, \times $ and $/$ accordingly. The tokenizer considers such tokens to be neither the end-of-file, words, nor numbers. As a result you can compare it against the character ‘$+’, ‘$-’, ‘$\times ’, and ‘$/$’. See the attached sample code.

Notes:
- To create a Character object from a char c, use new Character(c)
- The charValue() method of Character returns the char held by the Character object.
- For a stack, use the LRSSStackFactory from our original (not Prob. 1 above!) RAC framework (see attached page following the example code).

Write your solution on the next page
package algos;
import java.io.*;
import rac.*;

public class In2Post {

    private boolean isOp(int c) {
        return '+' == c || '-' == c || '*' == c || '/' == c;
    }

    /**
     * return true iff op1 has higher precedence than op2.
     */
    private boolean hasHigherPrecedence(char op1, char op2) {
        switch(op2) {
            case '+': case '-': return '*' == op1 || '/' == op1;
            case '*': case '/': return false;
            default: return true;
        }
    }

    private void createReadStream(String inFile, String outFile) {
        try {
            Reader fileReader;
            Writer fileWriter;
            PrintWriter printWriter;
            // Create a Reader object that reads from an input file named inFile:
            fileReader = new FileReader(inFile);
            // Create a Writer object that writes to an output file named outFile:
            fileWriter = new FileWriter(outFile);
            // Create a PrintWriter object to print characters to _fileWriter:
            printWriter = new PrintWriter(fileWriter);
            infix2postfix(fileReader, printWriter);
            printWriter.close(); // IMPORTANT!
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    public void infix2postfix(Reader fileReader, PrintWriter printWriter) {
        try {
            // Create a StreamTokenizer object to tokenize fileReader:
            StreamTokenizer tokenizer = new StreamTokenizer(fileReader);
            IRAContainer stack = LRSStackFactory.Singleton.makeRAC();
            stack.put(new Character('?'));
            int nextTok = tokenizer.nextToken();
            while (tokenizer.TT_EOF != nextTok && '$' != nextTok) {
                if (tokenizer.TT_NUMBER == tokenizer.ttype) {
                    printWriter.print(tokenizer.nval + " ");
                }
                else if (tokenizer.TT_WORD == tokenizer.ttype) {
                    throw new RuntimeException("Illegal Input : " + nextTok);
                }
                else if (isOp(nextTok)) {
                    char top = ((Character)stack.peek()).charValue();
                    while (hasHigherPrecedence((char)nextTok, top)) {
                        printWriter.print (top + " ");
                        stack.get();
                        top = ((Character)stack.peek()).charValue();
                    }
                    stack.put(new Character((char)nextTok));
                }
                else {
                    throw new RuntimeException("Illegal Input : " + nextTok);
                }
                nextTok = tokenizer.nextToken();
            }
        }
    }

}
while (!stack.isEmpty()) {
    printWriter.print(stack.get() + " ");
}

} catch (Exception e) {
    System.err.println(e.getMessage());
}
"*/

/** Run In2Post with 2 command line arguments.
 * @param arg[0] the input file name.
 * @param arg[1] the output file name.
 */
public static void main(String arg[]) {
    new In2Post().createReadWriteStreams(arg[0], arg[1]);
}